Effective interactions and operators
TALENT 2017

Gustav R. Jansen

National Center for Computational Sciences
Oak Ridge National Laboratory

Trento, July, 2017

Content

- Notation and Motivation
- Effective interaction in a matrix representation
- Basic ideas of the valence space expansion.
- A coupled representation
- Coupled-cluster effective interaction in the sd-shell

Section 1

Notation and Motivation

Problem statement

We are looking at non-relativistic particles, so the solutions of the A-body system, is given by the A-body Schrödinger equation,

$$\hat{\mathrm{H}}_A|\Psi_A\rangle = E_A|\Psi_A\rangle$$

Manybody wavefunction

The wavefunction of the manybody system can be decomposed into a suitable manybody basis

$$|\Psi_A\rangle = \sum_i c_i |\Phi_i\rangle.$$

For fermions, these are Slater-determinants

$$\begin{aligned} |\Phi_i\rangle &= |\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_A}\rangle \\ &= \left(\prod_{j=1}^{A} a_{i_j}^\dagger\right)|0\rangle, \end{aligned}$$

Where $a^\dagger$ is a second quantized operator satisfying

$$a_p^\dagger|0\rangle = |\alpha_p\rangle \qquad a_p|\alpha_q\rangle = \left(a_p^\dagger\right)^\dagger|\alpha_q\rangle = \delta_{pq}|0\rangle$$

$$\left\{a_p, a_q^\dagger\right\} = \delta_{pq} \qquad \{a_p, a_q\} = \left\{a_p^\dagger, a_q^\dagger\right\} = 0$$

Manybody wavefunction

In the **x**-representation the Slater-determinant is written

$$\langle \mathbf{x}|\Phi_i\rangle = \frac{1}{\sqrt{A}}\sum_{n=1}^{A!}(-1)^{P_n}\hat{\mathrm{P}}_n\prod_{j=1}^{A}\phi_{i,n_j}(\mathbf{x}_j),$$

where

$$\phi_{i,k}(\mathbf{x}_j) = \langle \mathbf{x}_j|\alpha_{i_k}\rangle$$

are the solutions to a selected single particle problem

$$\hat{\mathrm{h}}\phi_k(\mathbf{x}) = \epsilon_k\phi_k(\mathbf{x}).$$

Manybody wavefunction

In the particle-hole formalism all quantities are expressed in relation to the reference state

$$|\Phi_0\rangle = |\alpha_1 \dots \alpha_A\rangle, \quad \alpha_1, \dots, \alpha_A \leq \alpha_F$$

The indices are partitioned according to their relation to the Fermi level

$$i, j, \dots \leq \alpha_F \qquad a, b, \dots > \alpha_F \qquad p, q, \dots : \text{any},$$

and the second quantized operators now satisfy

$$\left\{a_i, a_j^\dagger\right\} = \delta_{ij} \qquad \left\{a_a, a_b^\dagger\right\} = \delta_{ab}$$

$$a_i|\Phi_0\rangle = |\Phi_i\rangle \qquad a_a^\dagger|\Phi_0\rangle = |\Phi^a\rangle$$

$$a_i^\dagger|\Phi_0\rangle = 0 \qquad a_a|\Phi_0\rangle = 0$$

Manybody wavefunction

The manybody wavefunction can be expanded in a linear combination of particle-hole excitations, which is complete in agiven basis set

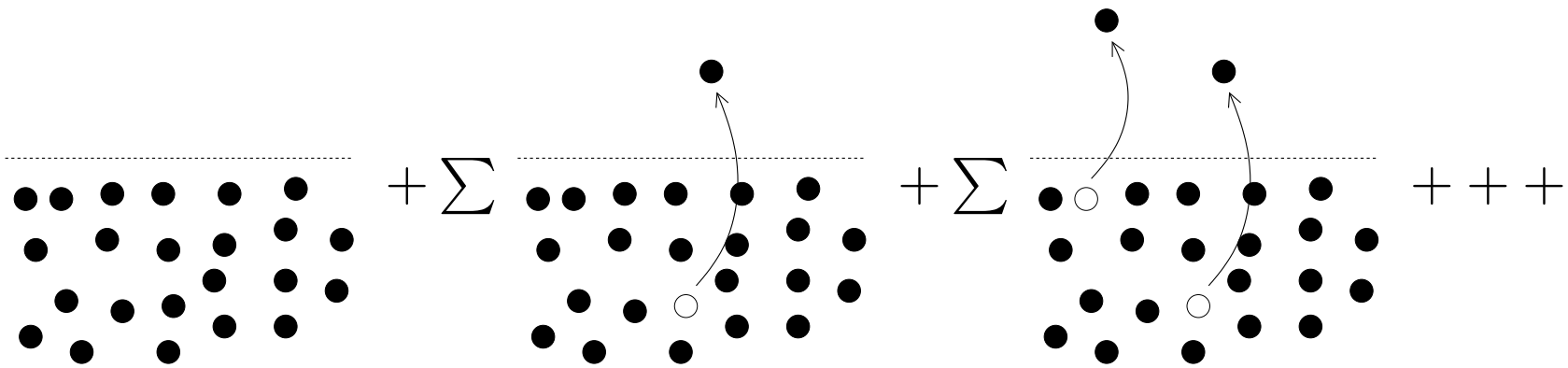
$$|\Psi\rangle = |\Phi_0\rangle + \sum_{ia} |\Phi_i^a\rangle + \frac{1}{4}\sum_{ijab} |\Phi_{ij}^{ab}\rangle + \ldots + \frac{1}{(A!)^2}\sum_{\substack{i_1\ldots i_A \\ a_1\ldots a_A}} |\Phi_{i_1\ldots i_A}^{a_1\ldots a_A}\rangle$$

$$= |\Phi_0\rangle + \sum_{ia} c_i^a a_a^\dagger a_i |\Phi_0\rangle + \frac{1}{4}\sum_{ijab} c_{ij}^{ab} a_a^\dagger a_b^\dagger a_j a_i |\Phi_0\rangle + \ldots +$$

$$\frac{1}{(A!)^2}\sum_{\substack{i_1\ldots i_A \\ a_1\ldots a_A}} c_{i_1\ldots i_A}^{a_1\ldots a_A} a_{a_1}^\dagger \ldots a_{a_A}^\dagger a_{i_A} \ldots a_{i_1} |\Phi_0\rangle$$

Manybody wavefunction

The particle-hole expansion of a manybody wavefunction is a linear combination of all possible excitations of the reference wavefuncton.



Manybody Hamiltonian

A general Hamiltonian contains up to A-body interactions

$$\hat{\mathrm{H}}_A = \sum_{i=1}^{A} \left(\hat{\mathrm{t}}_i + \hat{\mathrm{u}}_i\right) + \sum_{i<j=1}^{A} \hat{\mathrm{v}}_{ij} + + + \sum_{i_1<\cdots<i_A=1}^{A} \hat{\mathrm{v}}_{i_1,\ldots,i_A}$$

$$= \hat{\mathrm{T}}_{\mathrm{kin}} + \hat{\mathrm{U}} + \sum_{n=2}^{A} \hat{\mathrm{V}}_n,$$

where $\hat{\mathrm{T}}_{\mathrm{kin}}$ is the kinetic energy operator, $\hat{\mathrm{U}}$ is a generic onebody potential and $\hat{\mathrm{V}}_n$ is an n-body potential.

Manybody Hamiltonian

In second quantized form, a general n-body operator is written

$$\hat{\mathrm{V}}_n = \frac{1}{(n!)^2} \sum_{\substack{\alpha_1 \ldots \alpha_n \\ \gamma_1 \ldots \gamma_n}} \langle \alpha_1 \ldots \alpha_n | \hat{\mathrm{v}}_n | \gamma_1 \ldots \gamma_n \rangle a^\dagger_{\alpha_1} \ldots a^\dagger_{\alpha_n} a_{\gamma_n} \ldots a_{\gamma_1},$$

where the matrix elements $\langle \alpha_1 \ldots \alpha_n | \hat{\mathrm{V}}_n | \gamma_1 \ldots \gamma_n \rangle$ are fully anti-symmetric with respect to the interchange of indices and the sum over α_i and γ_i runs over all possible single particle states.

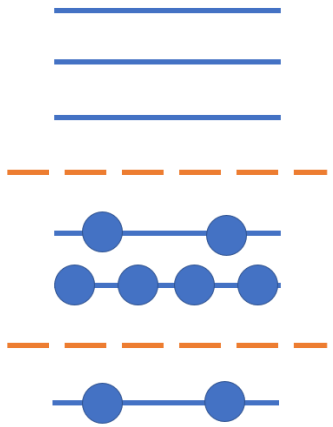
Manybody Hamiltonian

We will truncate the Hamiltonian at the $n = 2$ level and use a single particle basis diagonal in the onebody Hamiltonian, so the full Hamiltonian will be written

$$\hat{\mathrm{H}} = \sum_{pq} \langle p|\hat{\mathrm{h}}|q\rangle a_p^\dagger a_q + \frac{1}{4} \sum_{pqrs} \langle pq|\hat{\mathrm{v}}|rs\rangle a_p^\dagger a_q^\dagger a_s a_r$$

Counting Slater determinants

Because of quantum mechanics, a particle can exist in one of n single-particle levels.



Because nucleons are fermions, Pauli's exclusion principle applies. The number of possible ways to place k fermions in n levels is the same as picking k unordered outcomes from n possibilities, which is $\binom{n}{k}$. The total number of Slater determinants (N) become

$$N = \binom{n}{n_p} \times \binom{n}{n_n}$$

Counting Slater determinants

Jupyter notebook:nslater

```
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
rc("text", usetex=True)

def binomial(n, k):
    uvector = np.array(range(k+1, n+1))
    lvector = np.array(range(1, n-k+1))

    return np.e**(np.log(uvector).sum() - np.log(lvector).sum())
```

Counting Slater determinants

Jupyter notebook:nslater

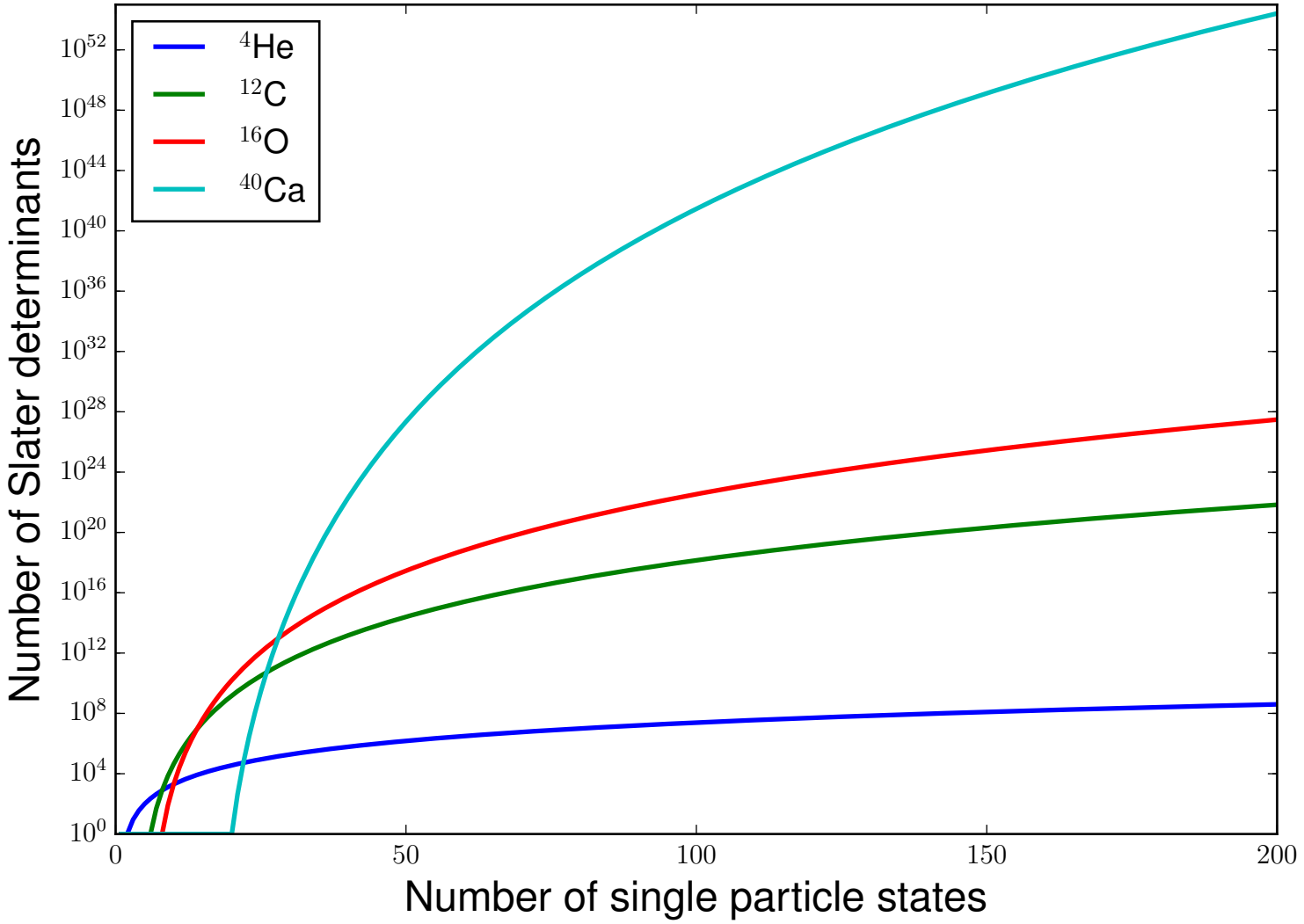
```
he4 = (2, 2)
c12 = (6,6)
o16 = (8,8)
ca40 = (20,20)
nuclei = [he4, c12, o16, ca40]
labels = [r"${}^4$He", r"${}^{12}$C", r"${}^{16}$O",
    r"${}^{40}$Ca"]
```

Counting Slater determinants

Jupyter notebook:nslater

```
x = range(1, 201)
for label, nucleus in zip(labels, nuclei):
    p = np.array([binomial(n, nucleus[0]) for n in x])
    n = np.array([binomial(n, nucleus[1]) for n in x])
    y = p*n
    plt.plot(x, y, linewidth=2, label=label)
plt.xlabel("Number of single particle states", fontsize=18)
plt.ylabel("Number of Slater determinants", fontsize=18)
plt.legend(loc=2)
plt.tight_layout()
plt.yscale("log")
plt.savefig("slater_determinants.pdf")
plt.show()
```

Curse of dimensionality



Approximations schemes

- Truncate the single particle basis.
- Truncate the set of possible Slater determinants.
  - Truncate by excitation level.
  - Truncate by total energy level.
  - Frozen core with valence space.
  - Coupled cluster truncation.

Section 2

Matrix representation

Eigenvalue systems

Definition

We define an eigenvalue problem of a real symmetric matrix $\mathbf{A}$ as

$$\mathbf{A}\mathbf{x}_j = \lambda_j \mathbf{x}_j,$$

where $\mathbf{A} = (a_{i,j}) \in \mathbb{R}^{n \times n}$, $\mathbf{x}_j = (x_{i,j}) \in \mathbb{R}^n$ is the j'th eigenvector of $\mathbf{A}$, and $\lambda_j \in \mathbb{R}$ is the j'th eigenvalue of $\mathbf{A}$.

Eigenvalue problem

Definition: EigenPair

```
import numpy as np

class EigenPair(object):
    def __init__(this, eigenvalue, eigenvector):
        this.eigenvalue = eigenvalue
        this.eigenvector = np.array(eigenvector)

    def __str__(this):
        s = "Eigenvalue: %s\n" % this.eigenvalue
        s += "Eigenvector: %s" % str(this.eigenvector)
        return s

    def is_eigenpair(this, matrix):
        return np.allclose(np.matmul(matrix,this.eigenvector),
            this.eigenvalue*this.eigenvector)
```

Eigenvalue problem

Definition: EigenSolver

```
import numpy as np
from eigenpair import EigenPair
class EigenSolver(object):
    def solve(this, matrix):
        w, v = np.linalg.eig(matrix)
        return [EigenPair(w[i], v[:,i]) for i in range(len(w))]
```

```
class EigensolverSymmetric(EigenSolver):
    def solve(this, matrix):
        w, v = np.linalg.eigh(matrix)
        return [EigenPair(w[i], v[:,i]) for i in range(len(w))]
```

Eigenvalue systems

Eigen decomposition

The eigenvectors of $\mathbf{A}$ form the columns of an orthogonal matrix $\mathbf{Q}$ that satisfies

$$\mathbf{Q}\mathbf{Q}^T = \mathbf{I},$$

where $\mathbf{Q} = (q_{i,j}) \in \mathbb{R}^{n \times n}$, $q_{i,j} = x_{i,j}$ and $\mathbf{I}$ is the identity matrix.

The eigenvalues of $\mathbf{A}$ form the entries of a diagonal matrix $\mathbf{D} = (d_{i,j}) \in \mathbb{R}^{n \times n}$, $d_{j,j} = \lambda_j$.

The eigen decomposition of $\mathbf{A}$ is written

$$\mathbf{A} = \mathbf{Q}\mathbf{D}\mathbf{Q}^T.$$

Eigenvalue systems

Eigen decomposition

When $\mathbf{A}$ is not symmetric, but still have real eigenvalues and eigenvectors, the eigenvectors are not orthogonal, but we can write the eigendecomposition as

$$\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1},$$

where $\mathbf{V} = (v_{i,j}) \in \mathbb{R}^{n \times n}$, $v_{i,j} = x_{i,j}$.

Eigenvalue problem

Definition: EigenDecompostion

```
from eigensolver import EigenSolver, EigensolverSymmetric
import numpy as np
class EigenDecomposition(object):
    def __init__(this):
        this.solver = EigenSolver()

    def decompose(this, matrix):
        pairs = this.solver.solve(matrix)

        shape = (len(pairs), len(pairs))
        D = np.zeros(shape)
        V = np.zeros(shape)
        D[np.diag_indices_from(D)] = np.array([p.eigenvalue for
            p in pairs])
        for i in range(len(pairs)): V[:,i] = pairs[i].eigenvector

        return D, V
```

Eigenvalue problem

Definition: MatrixFromEigenDecomposition

```
import numpy as np

class MatrixFromEigenDecomposition(object):
    def construct(this, D, V):
        return np.matmul(V, np.matmul(D, np.linalg.inv(V)))

class MatrixFromEigenDecompositionSymmetric(object):
    def construct(this, D, V):
        return np.matmul(V, np.matmul(D, V.transpose()))
```

Eigenvalue systems

Effective matrix

Let's now define a matrix $\mathbf{B} = (b_{i,j}) \in \mathbb{R}^{m \times m}$, where $m < n$. It satisifies the eigenvalue equation

$$\mathbf{B}\mathbf{y}_j = \delta_j \mathbf{y}_j,$$

where $\mathbf{y}_j = (y_{i,j}) \in \mathbb{R}^m$ is the j'th eigenvector of $\mathbf{B}$, and $\delta_j \in \mathbb{R}$ is the j'th eigenvalue of $\mathbf{B}$.

We don't know the elements of $\mathbf{B}$, but we want its eigenvalues to be a subset of the eigenvalues of $\mathbf{A}$.

What is the simplest matrix that satisfies this criteria?

Eigenvalue systems

Effective matrix

The simplest matrix that has the eigenvalues $\delta_j = \lambda_{k_j}, j \in [1, m]$ is

$$\mathbf{B} = \mathbf{E},$$

where $\mathbf{E} = (e_{i,j}) \in \mathbb{R}^{m \times m}$, $e_{j,j} = \delta_j = \lambda_{k_j}$.

Eigenvalue systems

Effective matrix

The next step is to use the eigenvectors of **A** determine the eigenvectors of **B** and then construct **B** through its eigen decomposition.

Let the eigenvalues of **B** be $\delta_j = \lambda_{k_j}, j \in [1, m]$ as before, and define the eigenvectors as $\mathbf{y}_j = (y_{i,j}), y_{i,j} = x_{i_k,j_k}, k \in [1, m]$. Construct the matrix **Y** from the eigenvectors of **B**, so that $\mathbf{Y} = (y_{i,j})$.

If **X** was an orthogonal matrix, is **Y** an orthogonal matrix?

Eigenvalue systems

Effective matrix

We can now construct $\mathbf{B}$ through its eigen decomposition

$$\mathbf{B} = \mathbf{Y}^{-1}\mathbf{E}\mathbf{Y}.$$

If $\mathbf{A}$ was a symmetric matrix, is $\mathbf{B}$ symmetric?

Eigenvalue problem

Definition: EigenProjection

```
class EigenProjection(object):
    def project(this, pairs, mapping):
        return [p.project(mapping) for p in pairs]
```

```
class EigenPair(object):
    def project(this, mapping):
        eigenvalue = this.eigenvalue
        eigenvector = this.eigenvector[mapping]
        return EigenPair(eigenvalue, eigenvector)
```

Effective matrix

Symmetric orthogonalization

Define a similarity transformation of $\mathbf{B}$ as

$$\mathbf{C} = \mathbf{S}^{-1/2}\mathbf{B}\mathbf{S}^{1/2},$$

where $\mathbf{S} = \mathbf{Y}^T\mathbf{Y}$ has positive eigenvalues, $\mathbf{Y}$ is the matrix that diagonalizes $\mathbf{B}$ defined before, and

$$\mathbf{S}^{1/2} = \mathbf{Q}^T\mathbf{D}^{1/2}\mathbf{Q}.$$

We define the square root of a matrix by its eigen decomposition and take the square root of the diagonal matrix of eigenvalues. $\mathbf{Q}$ is orthogonal since $\mathbf{S}$ is symmetric.

$$\mathbf{S}^T = (\mathbf{Y}^T\mathbf{Y})^T = \mathbf{Y}^T(\mathbf{Y}^T)^T = \mathbf{Y}^T\mathbf{Y} = \mathbf{S}$$

Effective matrix

Symmetric orthogonalization

This new matrix

$$\mathbf{C} = \mathbf{S}^{-1/2}\mathbf{B}\mathbf{S}^{1/2},$$

is symmetric and its eigenvectors are the closest possible to the original eigenvectors.

C is now a symmetric matrix that reproduces the original eigenvalues that was selected from **A** with eigenvectors that are the closest possible to the original eigenvectors in a reduced space.

This is exactly what we need to construct an effective shell-model Hamiltonian.

Summary

- Begin by selecting a subset of eigenvalues that will be the eigenvalues of an effective matrix.
- Then, Construct a matrix from its eigen decompostion using the selected eigenvalues and the projections of the selected eigenvectors.
- Perform a symmetric orthogonaliztion procedure to get a symmetric matrix that can be used in additional work.

Section 3

Basic ideas

Effective interactions: Basic ideas

Define a modelspace

Before we can construct the effective interaction as a matrix, we must choose a basis or a model space for the effective interaction. We will use the *sd*-shell as an example, consisting of the $0d_{5/2}$, $1s_{1/2}$, and $0d_{3/2}$ single-particle levels and with ^{16}O as a core.

Effective interactions: Basic ideas

A general effective interaction

Within a model space, the number of two-body states possible is fixed, the same with the number of three-body states and all the way up to how ever many particles the model space can accomodate. The *sd* shell can accomodate 12 protons and 12 neutrons, so it's possible to construct a 24-body state in this model space.

Definition

Let's define a general effective Hamiltonian $\tilde{\mathrm{H}}_{A_c+k}$ as:

$$\tilde{\mathrm{H}}_{A_c+k} = \tilde{\mathrm{V}}_0 + \tilde{\mathrm{V}}_1 + \tilde{\mathrm{V}}_2 + \ldots + \tilde{\mathrm{V}}_k,$$

where $\tilde{\mathrm{V}}_k$ a k-body interaction and A_c is the number of particles in the chosen core. We want this Hamiltonian to reproduce all energy levels that are possible for a k-body state in the model space.

Effective interactions: Basic ideas

An effective interaction for the core ($k = 0$)

There is only one state allowed for the $k = 0$ system and that is the ground state of the core of the valence space. The effective operator only has a scalar part,

$$\tilde{\mathrm{H}}_{A_c} = \tilde{\mathrm{V}}_0 = E_c,$$

where E_c is the core or vacuum energy. For the *sd*-shell this is just the binding energy of ^{16}O.

Since it is a scalar, we can leave it out of the operator as long as we remember that our energy levels are reported using the binding energy of the core as the vacuum.

Effective interactions: Basic ideas

A one-body effective interaction ($k = 1$)

For the $k = 1$ system, only the single-particle states defining the model space are allowed. The effective operator is now written as

$$\tilde{\mathrm{H}}_{A_c+1} = E_c + \tilde{\mathrm{V}}_1,$$

where we have kept the core energy just to make it clear that we are defining an operator that reproduce the binding energy of the core as well as the energy levels of the $A_c + 1$ system.

By knowing the total binding energy of the core and the single particle states, we can define a one-body effective operator for a model space. In fact, we could take these values from experimental data, actual calculations, or even use them to tune our effective interactions. In the literature, all these strategies have been and are being pursued.

Effective interactions: Basic ideas

A two-body effective interaction ($k = 2$)

The first non-trivial part of the effective interaction, is the two-body operator. We will write

$$\tilde{\mathrm{H}}_{A_c+2} = E_c + \tilde{\mathrm{V}}_1 + \tilde{\mathrm{V}}_2$$

for an effective interaction that reproduce the binding energy of the core, the single particle energies of the $A_c + 1$ system and the energy levels of all $A_c + 2$-body states allowed in our model space.

In theory, we could define this operator using only the $A_c + 2$-body energy levels as input. These could be taken from experimental data, from theory or they could be used as parameters in an optimization framework. Again, you will find a use for all these strategies in the literature.

Effective interactions: Basic ideas

A general effective interaction

By solving the Schrödinger equation for a k-body system using an effective interaction defined in a small model space, we can reproduce the energy levels for all $A_c + k$-body states allowed in the chosen model space.

Of course, the only thing we have done at this point is to create a model that reproduce all of our input data, which is not very useful. From a theoretical point of view, we would have to solve the Schrödinger equation for the full $A_c + k$-body system by some other method which defeates the purpose of this procedure.

Effective interactions: Basic ideas

A truncated effective interaction

The real power of this method is that we can truncate the effective interaction at the two-body level and solve $A_c + k$-body problems as k-body problems in small model spaces with only knowledge of relevant A_c, $A_c + 1$, and $A_c + 2$ solutions.

Effective interactions: Basic ideas

Onebody operator in a two-body basis

To construct the two-body interaction, we need to subtract off the one-body operator from the effective Hamiltonian. To do that we need to find expressions for a one-body operator in a two-body basis.

$$\langle pq|\tilde{V}_1|rs\rangle = \langle pq| \sum_{\alpha\beta} \langle \alpha|\tilde{v}_1|\beta\rangle a^\dagger_\alpha a_\beta |rs\rangle$$

$$= \sum_{\alpha\beta} \langle \alpha|\tilde{v}_1|\beta\rangle \langle 0|a_q a_p a^\dagger_\alpha a_\beta a^\dagger_r a^\dagger_s|0\rangle$$

To get the analytic expression we could use the anticommutation relations to normal order the operator string and find all the fully contracted terms.

Effective interactions: Basic ideas

Onebody operator in a two-body basis

Let's instead use Wick's theorem to find the fully contracted terms.

$$\langle pq|\tilde{\mathrm{V}}_1|rs\rangle = \sum_{\alpha\beta}\langle\alpha|\tilde{\mathrm{v}}_1|\beta\rangle\langle 0|$$
$$\Big(\overbrace{a_q a_p a_\alpha^\dagger a_\beta a_r^\dagger a_s^\dagger} + \overbrace{a_q a_p a_\alpha^\dagger a_\beta a_r^\dagger a_s^\dagger}$$
$$+ \overbrace{a_q a_p a_\alpha^\dagger a_\beta a_r^\dagger a_s^\dagger} + \overbrace{a_q a_p a_\alpha^\dagger a_\beta a_r^\dagger a_s^\dagger}\Big)|0\rangle$$

Effective interactions: Basic ideas

Onebody operator in a two-body basis

And then find the delta functions and insert into the sum over α and β

$$\langle pq|\tilde{\mathrm{V}}_1|rs\rangle = \sum_{\alpha\beta}\langle\alpha|\tilde{\mathrm{v}}_1|\beta\rangle\Big(\delta_{s\beta}\delta_{rp}\delta_{\alpha q} - \delta_{r\beta}\delta_{sp}\delta_{\alpha q} - \delta_{s\beta}\delta_{rq}\delta_{\alpha p} + \delta_{r\beta}\delta_{sq}\delta_{\alpha p}\Big)$$

$$= \langle p|\tilde{\mathrm{v}}_1|r\rangle\delta_{sq} + \langle q|\tilde{\mathrm{v}}_1|s\rangle\delta_{rp} - \langle p|\tilde{\mathrm{v}}_1|s\rangle\delta_{rq} - \langle q|\tilde{\mathrm{v}}_1|r\rangle\delta_{sp}$$

Section 4

A coupled representation

Angular momentum of a two-body state

Notation

We label a state with total angular momentum $\mathbf{j}$ and projection m as $|jm\rangle$ or $\langle jm|$. Similarily a two-body state is labelled by two angular momentum quantum numbers $\mathbf{j}_1$ and $\mathbf{j}_2$ and their projections m_1 and m_2 and write $|j_1 m_1 j_2 m_2\rangle$ or $\langle j_1 m_1 j_2 m_2|$.

The Wigner–Eckart theorem

We define the reduced matrix element of the m'th component of a spherical tensor T of rank j, by the Wigner-Eckart theorem.

$$\langle j_1 m_1 | \hat{\mathrm{T}}^j_m | j_2 m_2 \rangle = \langle j m j_2 m_2 | j_1 m_1 \rangle \langle j_1 \| \hat{\mathrm{T}}^j \| j_2 \rangle$$

For the special case where $j = m = 0$, we find

$$\langle j_1 m_1 | \hat{\mathrm{T}}^0_0 | j_2 m_2 \rangle = \delta_{j_1, j_2} \delta_{m_1, m_2} \langle j_1 \| \hat{\mathrm{T}}^0 \| j_2 \rangle,$$

since $\langle 00 j_2 m_2 | j_1 m_1 \rangle = \delta_{j_1, j_2} \delta_{m_1, m_2}$.

Angular momentum coupled scheme

Definition: Two-body matrix element

$$\langle pq; j_p m_p j_q m_q | \hat{\mathrm{H}} | rs; j_r m_r j_s m_s \rangle = \sum_{jm} \langle j_p m_p j_q m_q | jm \rangle \langle j_r m_r j_s m_s | jm \rangle \times \langle pq; j_p j_q; j \| \hat{\mathrm{H}} \| rs; j_r j_s; j \rangle$$

$$\langle pq; j_p j_q; j \| \hat{\mathrm{H}} \| rs; j_r j_s; j \rangle = \frac{1}{2j+1} \sum_{\substack{m_p m_q \\ m_r m_s m}} \langle j_p m_p j_q m_q | jm \rangle \langle j_r m_r j_s m_s | jm \rangle \times \langle pq; j_p m_p j_q m_q | \hat{\mathrm{H}} | rs; j_r m_r j_s m_s \rangle$$

Angular momentum coupled basis

Uncoupled to coupled derivation

We start with the uncoupled matrix element

$$\langle pq; j_p m_p j_q m_q | \hat{\mathrm{H}} | rs; j_r m_r j_s m_s \rangle,$$

where j_i and m_i identifies the angular momentum and its projection of the particle labelled with the index i, while the i identifies all other quantum numbers relevant for the state. $\hat{\mathrm{H}}$ is the Hamiltonian and it is a scalar with respect to the total angular momentum, since total angular momentum is conserved.

Angular momentum coupled basis

Uncoupled to coupled derivation

We start by inserting the identity operator in the coupled basis defined by

$$\mathbb{1} = \sum_{j_{12} m_{12}} |j_1 j_2; j_{12} m_{12}\rangle\langle j_1 j_2; j_{12} m_{12}|$$

The sums are over the total angular momentum and projection. After insertion we have

$$\langle pq; j_p m_p j_q m_q|\hat{\mathrm{H}}|rs; j_r m_r j_s m_s\rangle = \sum_{\substack{j_{pq} m_{pq} \\ j_{rs} m_{rs}}} \langle j_p m_p j_q m_q|j_p j_q; j_{pq} m_{pq}\rangle$$

$$\langle pq; j_p j_q; j_{pq} m_{pq}|\hat{\mathrm{H}}|rs; j_r j_s; j_{rs} m_{rs}\rangle\langle j_r j_s; j_{rs} m_{rs}|j_r m_r j_s m_s\rangle$$

Angular momentum coupled basis

Uncoupled to coupled derivation

The last step is to use the Wigner-Eckart theorem to separate the m-dependence from the Hamiltonian matrix element. We write

$$\langle pq; j_p m_p j_q m_q | \hat{\mathrm{H}} | rs; j_r m_r j_s m_s \rangle = \sum_{\substack{j_{pq} m_{pq} \\ j_{rs} m_{rs}}} \langle j_p m_p j_q m_q | j_{pq} m_{pq} \rangle$$
$$\langle j_r m_r j_s m_s | j_{rs} m_{rs} \rangle \langle pq; j_p j_q; j_{pq} \| \hat{\mathrm{H}} \| rs; j_r j_s; j_{rs} \rangle \delta_{j_{pq}, j_{rs}} \delta_{m_{pq}, m_{rs}},$$

where

$$\langle j_p m_p j_q m_q | j_{pq} m_{pq} \rangle \equiv \langle j_p m_p j_q m_q | j_p j_q; j_{pq} m_{pq} \rangle$$

are Clebsh-Gordon coefficients and

$$\langle j_p m_p j_q m_q | j_p j_q; j_{pq} m_{pq} \rangle = \langle j_p j_q; j_{pq} m_{pq} | j_p m_p j_q m_q \rangle.$$

Angular momentum coupled basis

Uncoupled to coupled derivation

Finally, we get

$$\langle pq; j_p m_p j_q m_q | \hat{\mathrm{H}} | rs; j_r m_r j_s m_s \rangle = \sum_{jm} \langle j_p m_p j_q m_q | jm \rangle \langle j_r m_r j_s m_s | jm \rangle \langle pq; j_p j_q; j \| \hat{\mathrm{H}} \| rs; j_r j_s; j \rangle .$$

Angular momentum coupled basis

Coupled to uncoupled derivation

Reversely, we start with the coupled matrix element

$$\langle pq; j_p j_q; j \| \hat{\mathrm{H}} \| rs; j_r j_s; j \rangle,$$

where both the bra an ket state are coupled to total angular momentum j.

Angular momentum coupled basis

Coupled to uncoupled derivation

We now start with the Wigner-Eckart theorem in reverse. We write

$$\langle pq; j_p j_q; j \| \hat{\mathrm{H}} \| rs; j_r j_s; j \rangle = \frac{1}{2j+1} \sum_m \langle pq; j_p j_q; jm | \hat{\mathrm{H}} | rs; j_r j_s; jm \rangle$$

since the Hamiltonian is a scalar with respect to angular momentum and the sum goes from $m = -j$ to $m = j$..

Angular momentum coupled basis

Coupled to uncoupled derivation

Now we insert the identity map, this time in the uncoupled basis defined by

$$\mathbb{1} = \sum_{m_1 m_2} |j_1 m_1 j_2 m_2\rangle\langle j_1 m_1 j_2 m_2|$$

The sums are now over the individual angular momentum projections that are missing from the coupled basis. After insertion we have

$$\langle pq; j_p j_q; j \| \hat{\mathrm{H}} \| rs; j_r j_s; j \rangle = \frac{1}{2j+1} \sum_{\substack{m_p m_q \\ m_r m_s m}} \langle j_p j_q; jm | j_p m_p j_q m_q \rangle \langle pq; j_p m_p j_q m_q | \hat{\mathrm{H}} | rs; j_r m_r j_s m_s \rangle \langle j_r m_r j_s m_s | j_r j_s; jm \rangle$$

where we identify the transformation coefficients as Clebsh-Gordon coefficients.

Angular momentum coupled basis

Coupled to uncoupled derivation

Finally we have

$$\langle pq; j_p j_q; j \| \hat{\mathrm{H}} \| rs; j_r j_s; j \rangle = \frac{1}{2j+1} \sum_{\substack{m_p m_q \\ m_r m_s m}} \langle j_p m_p j_q m_q | jm \rangle \langle j_r m_r j_s m_s | jm \rangle \langle pq; j_p m_p j_q m_q | \hat{\mathrm{H}} | rs; j_r m_r j_s m_s \rangle .$$

Angular momentum coupled basis

Consequences

Using the angular momentum coupled basis has a few major consequences.

- The two-body basis no longer depends on the projection of angular momentum. That means that the single particle basis is reduced in size.
- The Hamiltonian is diagonal in total angular momentum and independent of its projection. The result is that the storage required for the Hamiltonian is also reduced.
- All equations and expressions have to be rewritten in the coupled basis, which is a non-trivial task.
- Antisymmetry is hard(er) to express.

Angular momentum coupled basis

Single particle basis

We can now define a single particle basis with the quantum numbers n, l, j, and t_z where

- n is the number of nodes in the wavefunction,
- l is the orbital momentum,
- j is the total angular momentum ($\mathbf{j} = \mathbf{l} + \mathbf{s}$),
- t_z is the isospin projection ($t_z = -1/2$ for protons, $t_z = 1/2$ for neutrons)..

Angular momentum coupled basis

Definition: SingleParticleState

```
class SingleParticleState(object):
    def __init__(this, n, l, j, tz):
        this.pw = PartialWave(n, l, j)
        this.isospin =
            SingleParticleState.isospin_projection[int(tz)]

    def __str__(this):
        return "%s: %s" % (this.get_isospin_symbol(),
            str(this.pw))

    def __eq__(this, other):
        return this.pw == other.pw and this.isospin ==
            other.isospin

    def __neq__(this, other): return not this == other
```

Angular momentum coupled basis

Definition: SingleParticleState

```
class SingleParticleState(object):
    def get_isospin_symbol(this): return this.isospin
    def get_parity(this): return this.pw.get_parity()
    def get_isospin(this): return
        this.isospin_projection_reverse[this.isospin]
    def get_spin(this): return this.pw.get_spin()
```

Angular momentum coupled basis

Definition: PartialWave

```
class PartialWave(object):
    def __init__(this, n, l, j):
        this.n = int(n); this.l = int(l); this.j = int(j)

    def get_parity(this): return (-1)**this.l
    def get_spin(this): return this.j

    def __str__(this):
        return "%d%s_{%d/2}" % \
            ( this.n, this.spectral_notation_reverse[this.l],
                this.j)

    def __eq__(this, other):
        return this.n == other.n and this.l == other.l and
            this.j == other.j

    def __neq__(this, other): return not this == other
```

Angular momentum coupled basis

Definition: SingleParticleBasis

The SingleParticleBasis class is just a container for all SingleParticleState objects defining the model space.

```
class SingleParticleBasis(object):
    def __init__(this, states):
        this.states = states

    def __str__(this):
        s = ""
        for state in this.states:
            s += str(state) + "\n"
        return s[:-1]
```

Angular momentum coupled basis

Definition: SingleParticleBasisCreator

```
class SingleParticleBasisCreator(object):
    sdshell_partial_waves = [
        [ 0, 2, 5 ],
        [ 1, 0, 1 ],
        [ 0, 2, 3 ] ]

    @staticmethod
    def create_sd_shell(isospin=None):
        return SingleParticleBasisCreator.create_shell(
            SingleParticleBasisCreator.sdshell_partial_waves,
                isospin)
```

Angular momentum coupled basis

Definition: SingleParticleBasisCreator

```
class SingleParticleBasisCreator(object):
    @staticmethod
    def create_shell(partial_waves, isospin):
        if isospin is None: isospin = [-1, 1 ]
        else: isospin = [isospin]

        states = []
        for tz in isospin:
            for pw in partial_waves:
                n, l, j = pw
                states.append(SingleParticleState(n, l, j, tz))
        return SingleParticleBasis(states)
```

Angular momentum coupled basis

Usage: SingleParticleBasisCreator

```
print SingleParticleBasisCreator.create_sd_shell()
```

```
> python single_particle_basis_creator.py
proton: 0d_{5/2}
proton: 1s_{1/2}
proton: 0d_{3/2}
neutron: 0d_{5/2}
neutron: 1s_{1/2}
neutron: 0d_{3/2}
```

Angular momentum coupled basis

Channels

We can now exploit all the symmetries of the Hamiltonian and label diagonal blocks by the conserved quantum numbers isospin projection (t_z), parity (π), and total angular momentum (j).

We will call a diagonal block with a specific set of quantum numbers a channel.

Angular momentum coupled basis

Definition: TwobodyChannel

```
from parity import Parity
class TwobodyChannel(object):
    def __init__(this, spin, isospin, parity):
        this.spin = int(spin)
        this.isospin = int(isospin)
        this.parity = Parity(parity)

    def __str__(this):
        return "Spin: %d, Isospin: %s, Parity: %s" % \
      (this.spin, this.get_isospin_symbol(),
          this.get_parity_symbol())
```

Angular momentum coupled basis

Definition: TwobodyChannel

```
class TwobodyChannel(object):
    def get_spin(this): return this.spin

    def get_isospin(this): return this.isospin

    def get_parity(this): return this.parity.get_parity()

    def get_parity_symbol(this): return
        this.parity.get_parity_symbol()

    def get_isospin_symbol(this): return
        this.isospin_conversion[this.isospin]
```

Angular momentum coupled basis

Definition: TwobodyChannelCreator

```
class TwobodyChannelCreator(object):
  @staticmethod
  def create_channels(basis):
    channels = []
    parity_channels = TwobodyChannelCreator.find_parity(basis)
    isospin_channels = TwobodyChannelCreator.find_isospin(basis)
    spin_channels = TwobodyChannelCreator.find_spin(basis)
    for parity in parity_channels:
      for isospin in isospin_channels:
        for spin in spin_channels:
          channels.append(TwobodyChannel(spin, isospin, parity))

   return channels
```

Angular momentum coupled basis

Usage: TwobodyChannelCreator

```
from single_particle_basis_creator import
    SingleParticleBasisCreator

sd = SingleParticleBasisCreator.create_sd_shell()
channels = TwobodyChannelCreator.create_channels(sd)
for c in channels: print c
```

Angular momentum coupled basis

Usage: TwobodyChannelCreator

```
> python twobody_channel_creator.py
Spin: 0, Isospin: pp, Parity: +
Spin: 1, Isospin: pp, Parity: +
Spin: 2, Isospin: pp, Parity: +
Spin: 3, Isospin: pp, Parity: +
Spin: 4, Isospin: pp, Parity: +
Spin: 5, Isospin: pp, Parity: +
Spin: 0, Isospin: pn, Parity: +
Spin: 1, Isospin: pn, Parity: +
Spin: 2, Isospin: pn, Parity: +
Spin: 3, Isospin: pn, Parity: +
Spin: 4, Isospin: pn, Parity: +
Spin: 5, Isospin: pn, Parity: +
Spin: 0, Isospin: nn, Parity: +
Spin: 1, Isospin: nn, Parity: +
Spin: 2, Isospin: nn, Parity: +
Spin: 3, Isospin: nn, Parity: +
Spin: 4, Isospin: nn, Parity: +
Spin: 5, Isospin: nn, Parity: +
```

Angular momentum coupled basis

Definition: TwobodyState

```
from single_particle_state import SingleParticleState
from parity import Parity

class TwobodyState(object):
    def __init__(this, a, b):
        if type(a) is SingleParticleState and type(b) is
            SingleParticleState:
            this.a = a
            this.b = b
        else: this.a = None; this.b = None

    def is_identical(this): return this.a == this.b

    def __str__(this):
        return "a: %s\nb: %s\n%s" %(str(this.a),
            str(this.b),this.is_identical())
```

Angular momentum coupled basis

Definition: TwobodyState

```
class TwobodyState(object):
  def get_parity_symbol(this):
    p = Parity(this.a.get_parity()*this.b.get_parity())
    return p.get_parity_symbol()

  def get_parity(this): return
      Parity(this.a.get_parity()*this.b.get_parity()).get_parity()

  def get_isospin(this): return (this.a.get_isospin() +
      this.b.get_isospin())/2

  def get_minimum_spin(this): return abs(this.a.get_spin() -
      this.b.get_spin())/2

  def get_maximum_spin(this): return (this.a.get_spin() +
      this.b.get_spin())/2
```

Angular momentum coupled basis

Definition: TwobodyBasis

```
from twobody_channel import TwobodyChannel

class TwobodyBasis(object):
    def __init__(this, channel, states):
        if type(channel) is TwobodyChannel: this.channel =
            channel
        this.states = states

    def __str__(this):
        s = str(this.channel) +"\n"
        for state in this.states:
            s += str(state) + "\n"
        return s[:-1]

    def get_basis_size(this): return len(this.states)
```

Angular momentum coupled basis

Definition: TwobodyBasisCreator

```
class TwobodyBasisCreator(object):
    def __init__(this, spbasis):
        this.spbasis = spbasis
```

Angular momentum coupled basis

Definition: TwobodyBasisCreator

```
class TwobodyBasisCreator(object):
  def get_basis_for_channel(this, ch):
    if type(channel) is not TwobodyChannel: return []

    states = []
    left = this.spbasis.states
    right = this.spbasis.states
    for a, idxa in zip(left, range(len(left))):
      for b, idxb in zip(right, range(len(right))):
        if idxb < idxa: continue
        s = TwobodyState(a, b)
        if s.get_parity() != ch.get_parity(): continue
        if s.get_isospin() != ch.get_isospin(): continue
        if ch.get_spin() < s.get_minimum_spin(): continue
        if ch.get_spin() > s.get_maximum_spin(): continue
        if a==b and ch.get_spin()%2 != 0: continue
        states.append(s)
    return TwobodyBasis(ch, states)
```

Angular momentum coupled basis

Usage: TwobodyBasisCreator

```
from twobody_channel_creator import TwobodyChannelCreator
from single_particle_basis_creator import
    SingleParticleBasisCreator as creator
from itertools import compress

spbasis = creator.create_sd_shell()
twobody_basis = TwobodyBasisCreator(spbasis)
channels = TwobodyChannelCreator.create_channels(spbasis)

print "Proton proton channels"
for ch in compress(channels, [ch.get_isospin() == -1 for ch
    in channels]):
   print twobody_basis.get_basis_for_channel(ch) + "\n"
```

Angular momentum coupled basis

Usage: TwobodyBasisCreator

```
> python twobody_basis_creator.py
Proton proton channels
Spin: 0, Isospin: pp, Parity: +
Number of states: 3
a: proton: 0d_{5/2}
b: proton: 0d_{5/2}
Identical: True
a: proton: 1s_{1/2}
b: proton: 1s_{1/2}
Identical: True
a: proton: 0d_{3/2}
b: proton: 0d_{3/2}
Identical: True

Spin: 1, Isospin: pp, Parity: +
Number of states: 2
a: proton: 0d_{5/2}
b: proton: 0d_{3/2}
Identical: False
a: proton: 1s_{1/2}
b: proton: 0d_{3/2}
Identical: False
```

Section 5

Example

Angular momentum coupled basis

Example: *sd*-shell

The *sd*-shell is built from three partial waves for protons and three for neutrons, namely the $0d_{5/2}$, $1s_{1/2}$ and $0d_{3/2}$ partial waves.

The two-body basis is split into three isospin projection channels, namely the proton-proton ($t_z = -1$), proton-neutron ($t_z = 0$), and neutron-neutron ($t_z = 1$) channels. In addition, only positive parity states are possible with this single particle basis.

Finally, we can create two-body states with total angular momentum $j \in [0, 5]$, with slight differences between the $t_z = \pm 1$ channels and the $t_z = 0$ channel because of the selection rule for identical particles.

Angular momentum coupled basis

Example: *sd*-shell

In this valence space we can create the following two-body states for identical particles:

- 0^+: $(0d_{5/2}, 0d_{5/2})$, $(1s_{1/2}, 1s_{1/2})$, and $(0d_{3/2}, 0d_{3/2})$.
- 1^+: $(0d_{5/2}, 0d_{3/2})$ and $(1s_{1/2}, 0d_{3/2})$
- 2^+: $(0d_{5/2}, 0d_{5/2})$, $(0d_{3/2}, 0d_{3/2})$, $(0d_{5/2}, 0d_{3/2})$, $(1s_{1/2}, 0d_{3/2})$, and $(0d_{5/2}, 1s_{1/2})$
- 3^+: $(0d_{5/2}, 1s_{1/2})$ and $(0d_{5/2}, 0d_{3/2})$
- 4^+: $(0d_{5/2}, 0d_{5/2})$ and $(0d_{5/2}, 0d_{3/2})$

Angular momentum coupled basis

Example: *sd*-shell

Additional two-body states we can create from proton-neutron combinations (proton state always listed first):

- 0^+:
- 1^+: $(0d_{5/2}, 0d_{5/2})$, $(1s_{1/2}, 1s_{1/2})$, $(0d_{3/2}, 0d_{3/2})$, $(0d_{3/2}, 0d_{5/2})$, and $(0d_{3/2}, 1s_{1/2})$
- 2^+: $(0d_{3/2}, 0d_{5/2})$, $(0d_{3/2}, 1s_{1/2})$, and $(1s_{1/2}, 0d_{5/2})$
- 3^+: $(0d_{5/2}, 0d_{5/2})$, $(0d_{3/2}, 0d_{3/2})$, $(0d_{3/2}, 0d_{5/2})$, and $(1s_{1/2}, 0d_{5/2})$
- 4^+: $(0d_{3/2}, 0d_{5/2})$
- 5^+: $(0d_{5/2}, 0d_{5/2})$

See matrix-block.py for how to calculate the two-body basis for different channels in the *sd*-shell.